CSE-276C HW3

Zhenyu Wu | PID: A69030822

November 10, 2024

1 Question 1

1.1 Question 1.a

To solve $y(x)$ given $y(1) = -1$, we have:

$$(1-y)dy = x^{-2}dx$$

$$\int 1dy - \int ydy = \int x^{-2}dx$$

$$y - \frac{y^2}{2} + \frac{1}{x} + C = 0$$

Substitute $y(1) = -1$, we can find $C = \frac{1}{2}$, then

$$y - \frac{y^2}{2} + \frac{1}{x} + \frac{1}{2} = 0$$

$$y^2 - 2y - \frac{2}{x} - 1 = 0$$

Solve quadratic equation of y, we can have:

$$y = \frac{2 \pm \sqrt{8(\frac{1}{x}+1)}}{2}$$

$$y = 1 \pm \frac{\sqrt{8(\frac{1}{x}+1)}}{2}$$

Since $y(1) = -1$ with minus sign, we have

$$y(x) = 1 - \frac{\sqrt{8(\frac{1}{x}+1)}}{2}$$

For $y(0)$, the second term $\frac{1}{x}$ will divide by 0, which is undefined. The function $y(x)$ has a singularity at $x = 0$. Also the differential equation has singularity at boundary point $x = 0$ as well. This is a improper function we were mentioned

during lecture, might need other tools to solve.

When we take the limit of y(x) as x goes to zero from the right hand side, we have $-\infty$ as result, and I also plotted the graph for $y(x)$

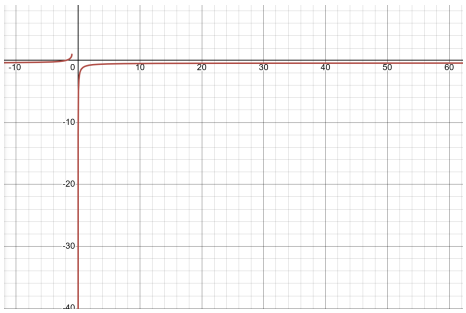

Figure 1: Analytics Solution

1.2 Question 1.b

For Euler Method, I implemented it as below:

```
def euler(x0, y, h, x):
    # Iterating till x
    while x0 < x:
        y = y + h * diffunc(x0, y)
        x0 = x0 + h
```

Listing 1: Euler Method

Each step, it would update both y and x starting from initial condition which is $y(1) = -1$ and make one step to ward left (toward 0) with step size of 0.05. The final estimation I got for $y(0) = -8.124934$ which is totally off from the $-\infty$, the result I had based on the limit. To further evaluate the accuracy, I checked $y(0.05)$, which is one step away from 0. The anaclitic solution gave $y_a(0.05) = -5.480740698$, and the Euler method gave $y_e(0.05) = -4.465853$, which was still not an accurate result.

1.3 Question 1.c

For the Runge-Kutta, I implemented as below.

```
def runge_kutta(x0, y, h, x):
    # Iterate til x
    while x0 > x:
```

```
        k_1 = h * diffunc(x0,y)
        k_2 = h * diffunc(x0 + (1/2) * h, y + (1/2) *
            k_1)
        k_3 = h * diffunc(x0 + (1/2) * h, y + (1/2) *
            k_2)
        k_4 = h * diffunc(x0 + h, y + k_3)
        y = y + (1/6) * k_1 + (1/3) * k_2 + (1/3) *
            k_3 + (1/6) * k_4
        x0 = x0 + h
```

Listing 2: Runge-Kutta

In every step going leftward from the right, it would compute 4 equations, k_1 to k_4, according to the equations from the slides. Then it would apply k_1 to k_4 to update y, and take one step forward for x with step size of 0.05.
The final estimation I got for $y(0) = -6.07173 \times 10^{27}$ (I only reported to 5 decimal places). Comparing to the Euler's Method, it gave a really large negative value and was closer from the $-\infty$, the result I had based on the limit. Therefore, I expected that it would have a better result for other x. To further evaluate the accuracy, I checked $y(0.05)$, which is one step away from 0. The anaclitic solution gave $y_a(0.05) = -5.480740698$, and the Runge-Kutta gave $y_r(0.05) = -5.500434$. Comparing to the Euler Method, the accuracy was improved a lot (from error > 1 to around 0.02).

1.4 Question 1.d

For the Bulirsch-Stoer method, the internal work flow of my implementation goes as follow:(I referenced both Numerical Recipes and other resources from Here)
For each step forward with step size $H = 0.05$, it will:

(1) Generate a list of substep size n using the strategy of $n = 2(j + 1)$

(2) For each substep size n, iterate from 1 to n and applied the modified mid-point strategy to get a $y(x + H)$ by

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x + mh, z_m) \quad \text{for } m = 1, 2, \ldots, n-1 \\ y(x + H) \approx y_n &\equiv \frac{1}{2}\left[z_n + z_{n-1} + hf(x + H, z_n)\right] \end{aligned}$$

Then stored $y(x + H)$ for each n.

(3) After went through some substep size n, apply the Richardson Extrapolation to refine estimation, which is

$$T_{k,j+1} = T_{kj} + \frac{T_{kj} - T_{k-1,j}}{\left(\frac{n_k}{n_{k-j}}\right)^2 - 1} \quad j = 0, 1, \ldots, k-1$$

(4) Determine the error

$$\text{err} = \|T_{kk} - T_{k,k-1}\|$$

If the error is below a specified error threshold value, then goes to next big step H and go back to step 1. If not, continue with higher n.(I explicitly set a variable K_{max} to restrict the largest n value. If $n = K_{max}$ and the error is still high, it will just continue and print the message suggesting that it doesn't converge)
As for the result, with the $K_{max} = 8$ which means the substep size $= [2, 4, 6, 8, 10, 12, 14, 16]$, the value of $y(0.05)$ for Bulirsch-Stoer is $y_b(0.05) = -5.48074058058637$ while the analytic solution gives $y_a(0.05) = -5.480740698$. Comparing to Euler and Runge-Kutta, Bulirsch-Stoer gives a further accurate result with error around 1×10^{-7}. I further tried larger K_{max}, which allows more substep sizes and smaller substep sizes, but this doesn't improve the result. However, I read one post from stackoverflow which applies the power of 2 to generate substep sizes. I tried this one, and this gives a slightly better result with $y(0.05) = -5.480740682289848$ with error of 1×10^{-8}.

2 Question 2

2.1 Question 2.1 Empty Table

Since I tried multiple ways, I reported all the methods I tried (starting with those eventually failed). The Final Answer I got will be at the end of this subsection (page 7-8).
My first thought was to do a PCA and plane fitting to get the dominate plane. After I did PCA like below:

```
def PCA(data):
    # Normalize
    pc = data - np.mean(data, axis=0)
    # Covariance Matrix
    Q = np.dot(pc.T, pc) / (pc.shape[0] - 1)
    U, Sigma, Vt = np.linalg.svd(Q)
    # Eigenvectors & Eigenvalue
    eigenvectors = U
    eigenvalues = Sigma
    # Normal Vector of plane
    normal_vector = eigenvectors[:, np.argmin(
        eigenvalues)]
    # Principle Componenet
    largest_vector = eigenvectors[:, np.argmax(
        eigenvalues)]
    second_largest_vector = eigenvectors[:, np.argsort
        (eigenvalues)[-2]]
```

```
    principal_component = np.hstack((largest_vector[:,
        np.newaxis], second_largest_vector[:, np.
       newaxis]))
    return normal_vector, principal_component
```

Listing 3: 'PCA'

After PCA, simply find the plane parameter by taking the normal vector and the centroid point of the plane as:

$$f = ax + by + cz + d = 0$$

However, after I make a projection of data to the plane, I got this:

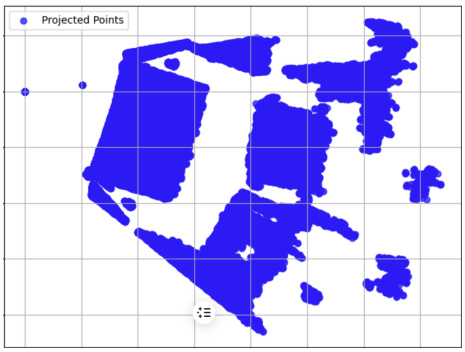

Figure 2: Raw Projection

This is the projection of all points onto the overall dominate plane of the whole points cloud. In order to find table, I tried to use KMean function to do the clustering, and eventually I got:

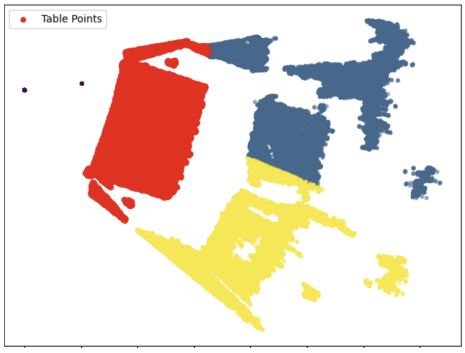

Figure 3: KMean and Projection

The red region is labeled as table which has the most dense points. However, since all other points were also included in the projection, this way failed. The

reason why I got this 2D projection includes: 1. Didn't filter out other irrelevant points, such as couches) 2. This is the overall dominate plane for the raw point cloud, which is not the plane for the table.
Move on to my second attempt. Since there are many points representing other surfaces or objects, I have a idea to use the RANSAC method to do multiple plane fitting for the whole point clouds. In the iteration, my implementation will call RANSAC to do a plane fitting by selecting random points, performing plane fitting, computing distance between other points and fitted plane to get inlier of the plane. Once it have inlier, just remove inlier from the original point cloud, and move on to next iteration. After iteration, just use the plane with the most dense inlier points. The result of RANSAC is(I only plotted the x,y of inlier)

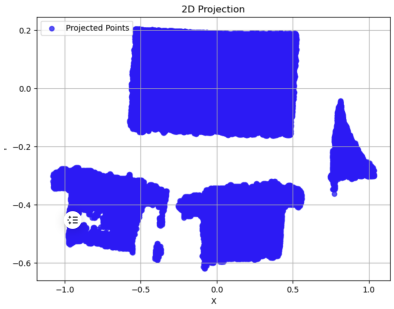

Figure 4: RANSAC inlier

Then, I applied the DBSCAN to cluster each component by given the radius of the neighborhood and minimum number of neighbors (data points) within radius, and then take the most dense cluster. In this way, I successfully find the table cluster, and the 2d projection of the table is

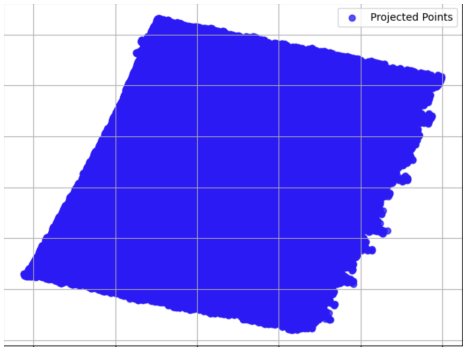

Figure 5: RANSAC Projection Table

The parameter I got is for this method

$$a = 0.0007352700619255972$$

$$b = 0.9034278478956234$$

$$c = 0.4287395281807107$$

$$d = -0.5995175646371099$$

However, I am not quite sure if the plane I got at the first place after computing RANSAC was the accurate table plane.Therefore, I came up with another method, which is my final method and my final answer.
The general idea of my final method is quite similar to RANSAC. In the iteration, my implementation will

1 Compute PCA and Plane Fitting with all available points

2 Compute both the distance difference and angle difference(normal vector of groups of close points) for all points and groups of points

3 Remove those points with distance and angle difference larger than specified threshold value

4 Dynamically adjust distance and angle difference threshold

At the beginning, since there were so many points in the point cloud, there were certain distance(error) between table plane and dominate plane of all points. Therefore, at the beginning of the iteration, the threshold would be quite loose, since we might also remove points for table given that error. After iterations, it would gradually throw away irrelevant points to reduce error and lower the chance of removing table points. Thus, my implementation would dynamically decrease the threshold value based on the convergence to make point classification more accurate. At the end of iteration, it saved the points for table , as well as some few scattered points that were in the table plane at the first place. Then I applied DBSCAN to do clustering and filter out all scattered points that were not part of table.
The final plot I have for the table plane is in Fig.6

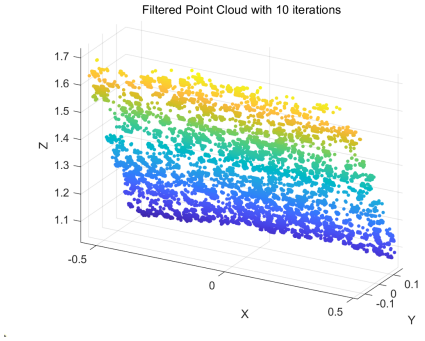

Figure 6: Final Plot for Empty Table

The final result for the table parameter is

$$a = 0.000323$$

$$b = 0.909455$$

$$c = 0.415802$$

$$d = -0.581354$$

2.2 Question 2.1 Clustered Table

For the clustered table, I applied the final method I mentioned for empty table. Since it will gradually remove irrelevant point that doesn't belong to the table plane. It could filter out the objects on the table. I tried two setting: 1. iterate 5 times, 2. iterate 10 times. The reason why I made two tests was because the points of table (especially the upper parts and parts that were occupied by object in the beginning) would become too sparse after points for objects being removed from the point clouds which would leave a hole. Plus the threshold value for the distance would gradually decrease, more and more table data would be accidentally removed with a really low distance threshold value(eg. 0.001). The figure for iterate 5 times is

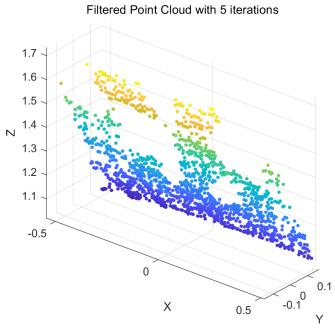

Figure 7: Clustered Table: iterate 5

The parameter I got is for this method

$$a = -0.004637$$

$$b = 0.906943$$

$$c = 0.421227$$

$$d = -0.584035$$

For this one, it kept more data points for the table, which might be helpful for estimating the table parameter. However, it only iterated for 5 times, the distance threshold value and error (between dominate plane and table plane) were both not low enough to remove points for top surface of the book on the top of the table (There were several(around 30 points) above and formed a tiny plane paralleled with the table plane). Like below

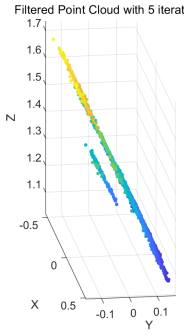

Figure 8: Clustered Table Side View: iterate 5

The 10 times iteration solved this problem, which means all the points for the objects and irreverent points were removed. However, the points for the table was quite sparse.

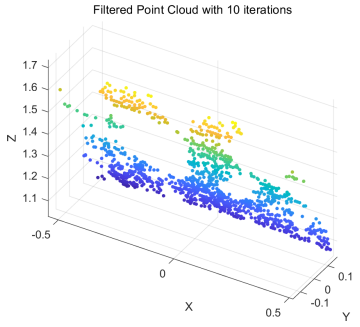

Figure 9: Clustered Table: iterate 10

The parameter I got for this one are

$$a = 0.001122$$

$$b = 0.910038$$

$$c = 0.414523$$

$$d = -0.580055$$

2.3 Question 2.2 CSE Building

My method for this problem is kind of combining previous methods. From the previous methods, I noticed that plane detection could be achieved through finding the normal vector of the plane, and use it to filter out outliers (based on distance and angle). Therefore, my method has following steps:

1 Use PCA and Plane Fitting to find the ground (dominate plane of raw data), which will generate the normal vector of ground, extract ground plane.

2 Based on the normal vector of ground, find normal vector of wall(should be perpendicular to ground normal vector). Plane Fitting with the wall normal vector, extract the most dense plane(the long side wall).

3 Once I have the normal vector of the long side wall and ground, compute the normal vectors that are perpendicular to both of them.

4 Plane fitting on those newly found normal vector, which should extract points all objects and other walls that on a plane perpendicular to ground and side wall.
After Step 1 and 2, I got this as result:

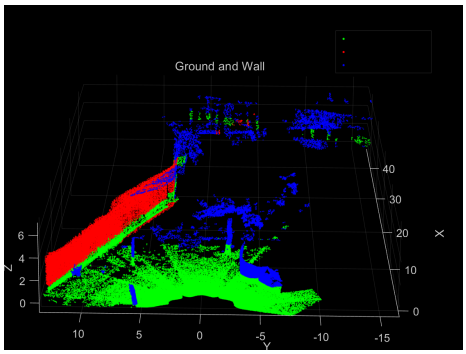

Figure 10: Long Side Wall and Ground

The Plane of ground is in green color, the side wall is in red color, and all other points that are not on either of them are in blue color. Then from this, I applied an iteration to do Step 3 and Step 4, since there are many objects and walls that are perpendicular to both ground and side wall. In each iteration, my implementation will 1.Compute normal vector(perpendicular to ground and side wall plane), 2. Do plane fitting and extract the plane with the most dense data, 3. Remove the extracted points and move on to find next plane. After iterations, it returned several plane for different objects, which shown below:

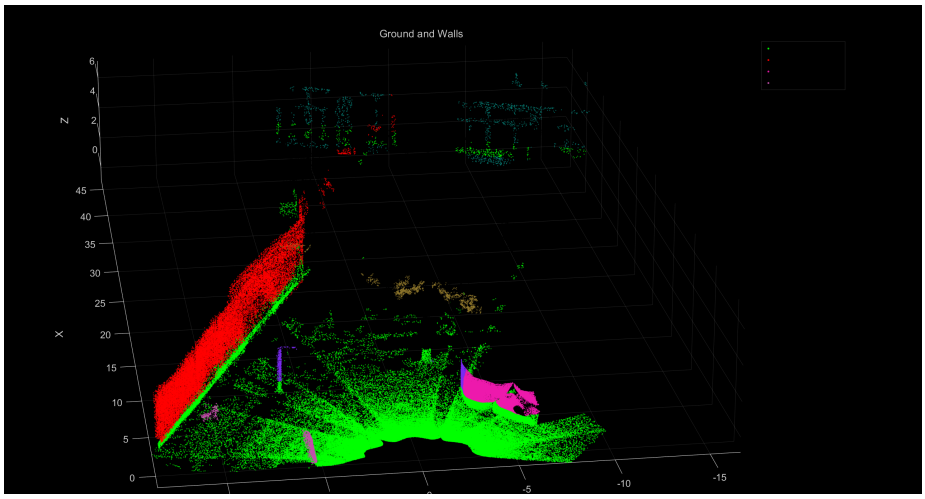

Figure 11: Found Plane

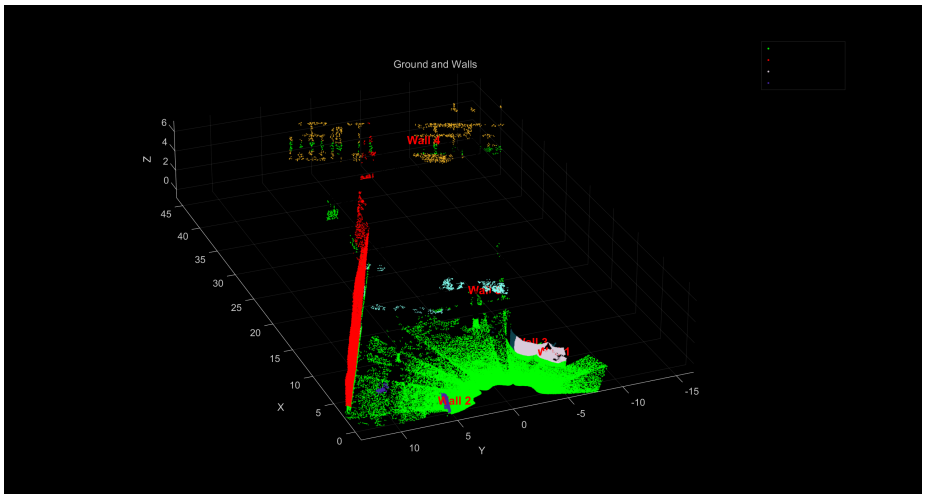

Figure 12: labeled Found Plane

I applied random color on each found plane, and in Fig.12, I labeled each found plane with text message. My implementation indeed found multiple planes that are perpendicular to the side wall and ground. The another plane of CSE building is included. Then as the result, the plane parameter for the long side wall (marked as red in Fig.11 and 12 are:

$$a = 0.3895$$

$$b = 0.9178$$

$$c = 0.0771$$

$$d = -13.0936$$

For another plane of CSE building, labled with dark green color(top corner) in Fig.11, and yellow color(top corner, with label "Wall 4") in Fig.12, the parameters are:

$$a = -0.9337$$

$$b = 0.3579$$

$$c = -0.0112$$

$$d = 42.7743$$